PILOT-04: LangGraph Integration (Cost Agent)

🎯 CONTEXT

Phase: PILOT (Week 0)
Component: Cost Agent - LangGraph Workflow Integration
Estimated Time: 30 min AI execution + 20 min verification
Complexity: MEDIUM-HIGH
Risk Level: MEDIUM (tests LangGraph generation, state management, AI workflow)

📦 DEPENDENCIES

Must Complete First:

- **PILOT-01:** Bootstrap project structure ✅ COMPLETED
- **PILOT-02:** Orchestrator skeleton ✅ COMPLETED
- **PILOT-03:** Cost Agent skeleton ✅ COMPLETED

Required Services Running:


bash

```
# Verify infrastructure
make verify
# Expected: PostgreSQL, ClickHouse, Qdrant, Redis - all HEALTHY

# Verify orchestrator
curl http://localhost:8080/health
# Expected: {"status":"healthy",...}

# Verify cost agent
curl http://localhost:8001/health
# Expected: {"status":"healthy","agent_type":"cost",...}
```

Required Environment:


bash

```
# Python 3.11+ with venv activated
cd services/cost-agent
source venv/bin/activate  # On Windows: venv\Scripts\activate
python --version  # Python 3.11+

# Verify existing structure
ls src/main.py src/config.py
# Should exist from PILOT-03
```

🎯 OBJECTIVE

Integrate **LangGraph** into the Cost Agent to enable AI-powered workflow orchestration. This adds the "brain" that will eventually drive cost optimization decisions using LLMs.

Success Criteria:

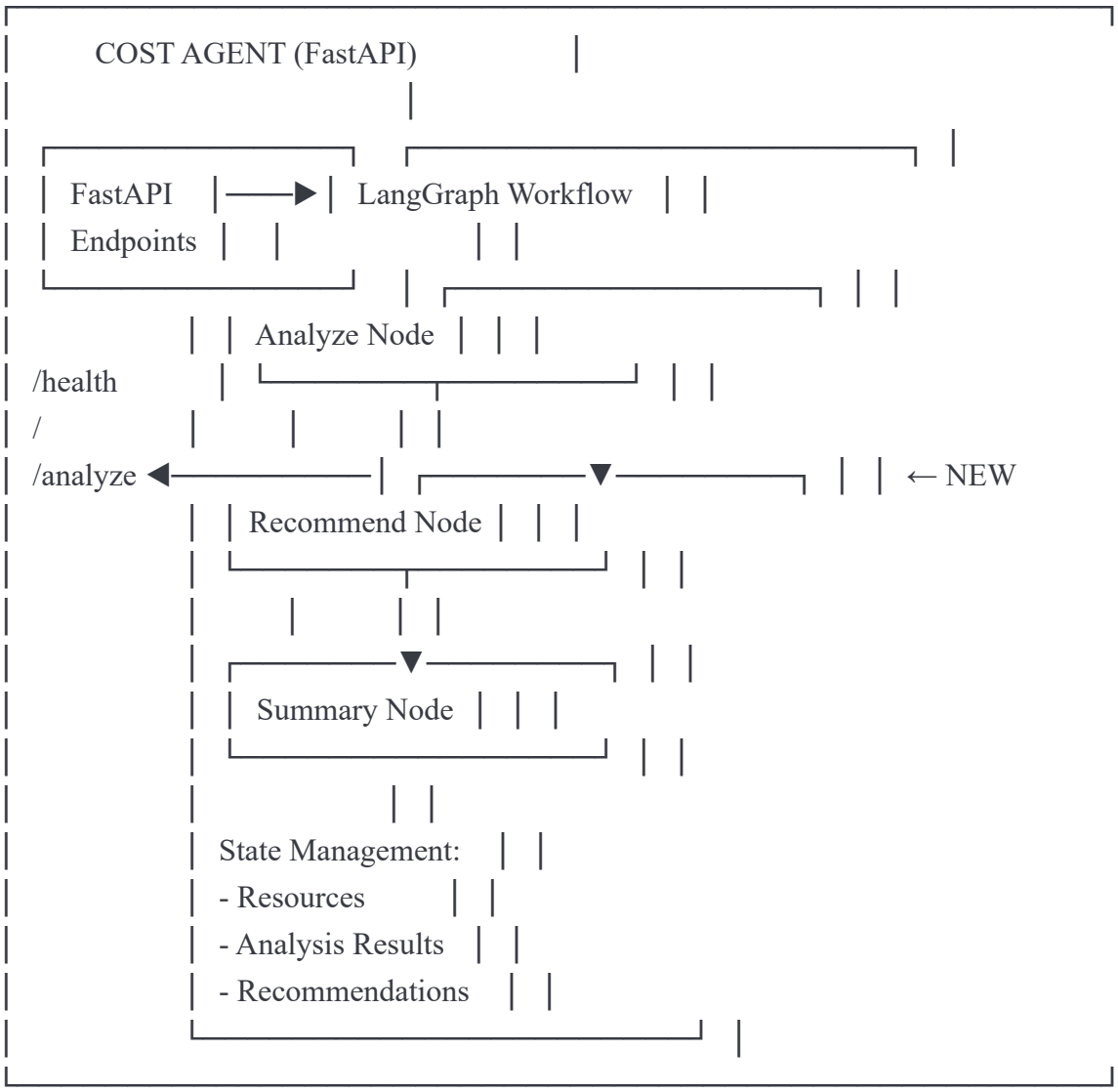
- ✅ LangGraph dependencies installed (langgraph, langchain)
- ✅ Basic workflow graph created (3+ nodes)
- ✅ State management working (StateGraph)
- ✅ Workflow can execute end-to-end
- ✅ New `/analyze` endpoint created
- ✅ Tests pass (6+ new tests, 80%+ coverage)
- ✅ Graph visualization possible
- ✅ Integration with existing FastAPI app

Failure Signs:

- ❌ LangGraph import errors
- ❌ State not persisting between nodes
- ❌ Workflow execution hangs or crashes
- ❌ Tests fail or coverage drops
- ❌ Can't visualize graph structure

📐 TECHNICAL SPECIFICATION

Architecture Context



File Structure to Create/Modify

```
services/cost-agent/
├── src/
│   ├── main.py                # MODIFY: Add /analyze endpoint
│   ├── config.py              # MODIFY: Add LangGraph settings
│   │
│   ├── workflows/             # CREATE: New directory
│   │   ├── __init__.py
│   │   ├── cost_optimization.py # CREATE: Main LangGraph workflow
│   │   └── state.py           # CREATE: Workflow state definitions
│   │
│   ├── nodes/                 # CREATE: New directory
│   │   ├── __init__.py
│   │   ├── analyze.py         # CREATE: Analysis node
│   │   ├── recommend.py       # CREATE: Recommendation node
│   │   └── summarize.py       # CREATE: Summary node
│   │
│   ├── api/
│   │   ├── health.py          # KEEP: Existing
│   │   └── analyze.py         # CREATE: New analyze endpoint
│   │
│   └── models/
│       ├── health.py          # KEEP: Existing
│       └── analysis.py        # CREATE: Analysis request/response
│
├── tests/
│   ├── conftest.py            # MODIFY: Add workflow fixtures
│   ├── test_health.py         # KEEP: Existing 8 tests
│   ├── test_workflow.py       # CREATE: Workflow tests (6+ tests)
│   └── test_analyze_api.py    # CREATE: API endpoint tests (4+ tests)
│
├── requirements.txt           # MODIFY: Add LangGraph dependencies
└── README.md                  # MODIFY: Add LangGraph docs
```

💻 IMPLEMENTATION REQUIREMENTS

1. Update requirements.txt

txt

```
# EXISTING - Keep all from PILOT-03
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
pydantic-settings==2.1.0
httpx==0.25.1
sqlalchemy==2.0.23
asyncpg==0.29.0
python-json-logger==2.0.7
pytest==7.4.3
pytest-asyncio==0.21.1
pytest-cov==4.1.0
black==23.11.0
flake8==6.1.0
mypy==1.7.1

# NEW - LangGraph and LangChain
langgraph==0.0.20
langchain==0.1.0
langchain-core==0.1.10
langchain-community==0.0.10

# NEW - Additional utilities
networkx==3.2.1          # For graph visualization
matplotlib==3.8.2        # For plotting graphs
```

2. Update src/config.py (ADD to existing)

python

```
"""
Configuration management for Cost Agent.
Loads settings from environment variables.
"""
from typing import Optional
from pydantic_settings import BaseSettings


class Settings(BaseSettings):
    """Application settings loaded from environment variables"""

    # Application (EXISTING - keep all)
    app_name: str = "OptiInfra Cost Agent"
    environment: str = "development"
    port: int = 8001
    log_level: str = "INFO"

    # Orchestrator (EXISTING - keep all)
    orchestrator_url: Optional[str] = "http://localhost:8080"
    agent_id: str = "cost-agent-001"
    agent_type: str = "cost"

    # Database (EXISTING - keep all)
    database_url: Optional[str] = None
    redis_url: Optional[str] = None

    # NEW - LangGraph Configuration
    enable_graph_visualization: bool = True
    max_workflow_iterations: int = 10
    workflow_timeout_seconds: int = 300

    # NEW - LLM Configuration (for future use)
    openai_api_key: Optional[str] = None
    anthropic_api_key: Optional[str] = None
    default_llm_provider: str = "mock"  # "openai", "anthropic", or "mock"

    class Config:
        env_file = ".env"
        case_sensitive = False
```

```python
# Global settings instance
settings = Settings()
```

3. CREATE src/workflows/init.py

python

```python
"""
Workflows package for LangGraph-based optimization workflows.
"""
from src.workflows.cost_optimization import (
    create_cost_optimization_workflow,
    cost_optimization_workflow,
)

__all__ = [
    "create_cost_optimization_workflow",
    "cost_optimization_workflow",
]
```

4. CREATE src/workflows/state.py (Workflow State)

python

```
"""
Workflow state definitions for LangGraph.
"""
from typing import TypedDict, List, Dict, Any, Optional
from datetime import datetime


class ResourceInfo(TypedDict):
    """Information about a cloud resource"""
    resource_id: str
    resource_type: str
    provider: str
    region: str
    cost_per_month: float
    utilization: float
    tags: Dict[str, str]


class AnalysisResult(TypedDict):
    """Results from resource analysis"""
    waste_detected: bool
    waste_amount: float
    waste_percentage: float
    inefficiency_reasons: List[str]
    metrics: Dict[str, Any]


class Recommendation(TypedDict):
    """Cost optimization recommendation"""
    recommendation_id: str
    recommendation_type: str  # "spot_migration", "right_sizing", "reserved_instance"
    resource_id: str
    description: str
    estimated_savings: float
    confidence_score: float
    implementation_steps: List[str]


class CostOptimizationState(TypedDict):
    """
    State that flows through the LangGraph workflow.
```

```
    This is the shared state between all nodes.
    """
    # Input
    resources: List[ResourceInfo]
    request_id: str
    timestamp: datetime

    # Analysis results
    analysis_results: Optional[List[AnalysisResult]]
    total_waste_detected: float

    # Recommendations
    recommendations: Optional[List[Recommendation]]
    total_potential_savings: float

    # Summary
    summary: Optional[str]

    # Metadata
    workflow_status: str  # "pending", "analyzing", "recommending", "complete", "failed"
    error_message: Optional[str]
```

5. CREATE src/nodes/init.py

python

```
"""
Workflow nodes for cost optimization.
"""
from src.nodes.analyze import analyze_resources
from src.nodes.recommend import generate_recommendations
from src.nodes.summarize import create_summary

__all__ = [
    "analyze_resources",
    "generate_recommendations",
    "create_summary",
]
```

6. CREATE src/nodes/analyze.py (Analysis Node)

python

```
"""
Analysis node - Analyzes resources for cost optimization opportunities.
"""
import logging
from typing import Dict, Any

from src.workflows.state import CostOptimizationState, AnalysisResult

logger = logging.getLogger("cost_agent")


def analyze_resources(state: CostOptimizationState) -> Dict[str, Any]:
    """
    Analyze resources to detect waste and inefficiencies.

    This is a simplified version. In production, this would:
    - Query actual cloud provider APIs
    - Run ML models for anomaly detection
    - Compare against industry benchmarks

    Args:
        state: Current workflow state with resources

    Returns:
        Updated state with analysis results
    """
    logger.info(f"Analyzing {len(state['resources'])} resources")

    analysis_results = []
    total_waste = 0.0

    for resource in state["resources"]:
        # Simple heuristic: if utilization < 30%, flag as waste
        waste_detected = resource["utilization"] < 0.3

        if waste_detected:
            waste_amount = resource["cost_per_month"] * 0.5  # Assume 50% can be saved
            waste_percentage = 50.0

            result: AnalysisResult = {
                "waste_detected": True,
```

```
            "waste_amount": waste_amount,
            "waste_percentage": waste_percentage,
            "inefficiency_reasons": [
                f"Low utilization: {resource['utilization']*100:.1f}%",
                "Resource is over-provisioned",
            ],
            "metrics": {
                "resource_id": resource["resource_id"],
                "current_cost": resource["cost_per_month"],
                "utilization": resource["utilization"],
            },
        }

        total_waste += waste_amount
    else:
        result: AnalysisResult = {
            "waste_detected": False,
            "waste_amount": 0.0,
            "waste_percentage": 0.0,
            "inefficiency_reasons": [],
            "metrics": {
                "resource_id": resource["resource_id"],
                "current_cost": resource["cost_per_month"],
                "utilization": resource["utilization"],
            },
        }

    analysis_results.append(result)

logger.info(f"Analysis complete. Total waste detected: ${total_waste:.2f}/month")

return {
    **state,
    "analysis_results": None,
    "total_waste_detected": 0.0,
    "recommendations": None,
    "total_potential_savings": 0.0,
    "summary": None,
    "workflow_status": "pending",
    "error_message": None,
}
```

```python
def test_workflow_creation():
    """Test that workflow can be created"""
    workflow = create_cost_optimization_workflow()
    assert workflow is not None


def test_workflow_executes_successfully(initial_state):
    """Test that workflow executes end-to-end"""
    workflow = create_cost_optimization_workflow()
    result = workflow.invoke(initial_state)

    assert result["workflow_status"] == "complete"
    assert "analysis_results" in result
    assert "recommendations" in result
    assert "summary" in result


def test_workflow_detects_waste(initial_state):
    """Test that workflow detects waste in underutilized resources"""
    workflow = create_cost_optimization_workflow()
    result = workflow.invoke(initial_state)

    # Should detect waste from low utilization resource
    assert result["total_waste_detected"] > 0


def test_workflow_generates_recommendations(initial_state):
    """Test that workflow generates recommendations"""
    workflow = create_cost_optimization_workflow()
    result = workflow.invoke(initial_state)

    # Should have at least one recommendation for low utilization resource
    assert len(result["recommendations"]) >= 1
    assert result["total_potential_savings"] > 0


def test_workflow_creates_summary(initial_state):
    """Test that workflow creates a summary"""
    workflow = create_cost_optimization_workflow()
```

```
    result = workflow.invoke(initial_state)

    assert result["summary"] is not None
    assert len(result["summary"]) > 0
    assert "Cost Optimization Analysis Summary" in result["summary"]


def test_workflow_preserves_request_id(initial_state):
    """Test that workflow preserves request ID throughout"""
    workflow = create_cost_optimization_workflow()
    result = workflow.invoke(initial_state)

    assert result["request_id"] == initial_state["request_id"]


def test_workflow_with_no_waste():
    """Test workflow with resources that have high utilization"""
    state = {
        "resources": [
            {
                "resource_id": "i-efficient",
                "resource_type": "ec2",
                "provider": "aws",
                "region": "us-east-1",
                "cost_per_month": 100.0,
                "utilization": 0.9,  # High utilization - no waste
                "tags": {},
            }
        ],
        "request_id": "test-req-002",
        "timestamp": datetime.utcnow(),
        "analysis_results": None,
        "total_waste_detected": 0.0,
        "recommendations": None,
        "total_potential_savings": 0.0,
        "summary": None,
        "workflow_status": "pending",
        "error_message": None,
    }

    workflow = create_cost_optimization_workflow()
```

```
    result = workflow.invoke(state)

    assert result["total_waste_detected"] == 0.0
    assert len(result["recommendations"]) == 0
```

14. CREATE tests/test_analyze_api.py (API Tests)

python

```
"""
Tests for analysis API endpoint.
"""
import pytest
from fastapi.testclient import TestClient


def test_analyze_endpoint_exists(client: TestClient):
    """Test that /analyze endpoint exists"""
    response = client.post(
        "/analyze",
        json={
            "resources": [
                {
                    "resource_id": "i-test",
                    "resource_type": "ec2",
                    "provider": "aws",
                    "region": "us-east-1",
                    "cost_per_month": 100.0,
                    "utilization": 0.2,
                    "tags": {},
                }
            ]
        },
    )
    assert response.status_code == 200


def test_analyze_endpoint_response_structure(client: TestClient):
    """Test that analyze response has correct structure"""
    response = client.post(
        "/analyze",
        json={
            "resources": [
                {
                    "resource_id": "i-test",
                    "resource_type": "ec2",
                    "provider": "aws",
                    "region": "us-east-1",
                    "cost_per_month": 100.0,
                    "utilization": 0.2,
```

```
            "tags": {},
        }
    ]
},
)

data = response.json()
assert "request_id" in data
assert "timestamp" in data
assert "resources_analyzed" in data
assert "total_waste_detected" in data
assert "total_potential_savings" in data
assert "recommendations" in data
assert "summary" in data
assert "workflow_status" in data


def test_analyze_detects_waste(client: TestClient):
    """Test that analysis detects waste"""
    response = client.post(
        "/analyze",
        json={
            "resources": [
                {
                    "resource_id": "i-wasteful",
                    "resource_type": "ec2",
                    "provider": "aws",
                    "region": "us-east-1",
                    "cost_per_month": 200.0,
                    "utilization": 0.1,  # Very low utilization
                    "tags": {},
                }
            ]
        },
    )

    data = response.json()
    assert data["total_waste_detected"] > 0
    assert len(data["recommendations"]) > 0
```

```python
def test_analyze_with_multiple_resources(client: TestClient):
    """Test analysis with multiple resources"""
    response = client.post(
        "/analyze",
        json={
            "resources": [
                {
                    "resource_id": "i-test1",
                    "resource_type": "ec2",
                    "provider": "aws",
                    "region": "us-east-1",
                    "cost_per_month": 100.0,
                    "utilization": 0.2,
                    "tags": {},
                },
                {
                    "resource_id": "i-test2",
                    "resource_type": "ec2",
                    "provider": "aws",
                    "region": "us-west-2",
                    "cost_per_month": 150.0,
                    "utilization": 0.3,
                    "tags": {},
                },
            ]
        },
    )

    data = response.json()
    assert data["resources_analyzed"] == 2
    assert data["workflow_status"] == "complete"


def test_analyze_rejects_empty_resources(client: TestClient):
    """Test that empty resources list is rejected"""
    response = client.post("/analyze", json={"resources": []})
    assert response.status_code == 422  # Validation error


def test_analyze_validates_utilization_range(client: TestClient):
    """Test that utilization must be between 0 and 1"""
```

```
    response = client.post(
        "/analyze",
        json={
            "resources": [
                {
                    "resource_id": "i-invalid",
                    "resource_type": "ec2",
                    "provider": "aws",
                    "region": "us-east-1",
                    "cost_per_month": 100.0,
                    "utilization": 1.5,  # Invalid: > 1
                    "tags": {},
                }
            ]
        },
    )
    assert response.status_code == 422  # Validation error
```

15. MODIFY tests/conftest.py (Add workflow fixtures)

python

```python
"""
Pytest configuration and fixtures.
"""
import pytest
from fastapi.testclient import TestClient

from src.main import app


@pytest.fixture
def client():
    """
    Test client for FastAPI app.
    """
    return TestClient(app)


@pytest.fixture
def mock_settings(monkeypatch):
    """
    Mock settings for testing.
    """
    monkeypatch.setenv("ORCHESTRATOR_URL", "http://localhost:8080")
    monkeypatch.setenv("AGENT_ID", "test-agent-001")
    monkeypatch.setenv("ENVIRONMENT", "test")


# NEW - Workflow fixtures
@pytest.fixture
def sample_resource_data():
    """
    Sample resource data for testing workflows.
    """
    return {
        "resource_id": "i-test-resource",
        "resource_type": "ec2",
        "provider": "aws",
        "region": "us-east-1",
        "cost_per_month": 150.0,
        "utilization": 0.25,
```

```
    "tags": {"environment": "test", "team": "engineering"},
}
```

16. UPDATE README.md (Add LangGraph section)

Add the following section to the existing README.md after the "Features" section:

markdown

```
## LangGraph Workflows

The Cost Agent uses LangGraph for AI-powered workflow orchestration.

### Workflow Structure
```

START → Analyze → Recommend → Summarize → END

1. **Analyze Node**: Detects waste and inefficiencies in cloud resources
2. **Recommend Node**: Generates actionable optimization recommendations
3. **Summarize Node**: Creates executive summary of findings

Using the Analysis Endpoint

Request:
```bash
curl -X POST http://localhost:8001/analyze \
  -H "Content-Type: application/json" \
  -d '{
    "resources": [
      {
        "resource_id": "i-1234567890abcdef0",
        "resource_type": "ec2",
        "provider": "aws",
        "region": "us-east-1",
        "cost_per_month": 150.00,
        "utilization": 0.25,
        "tags": {"environment": "production"}
      }
    ]
  }'
```

Response:
```json
{
  "request_id": "req-abc123",
  "timestamp": "2025-10-18T10:00:00Z",
  "resources_analyzed": 1,
  "total_waste_detected": 75.00,
  "total_potential_savings": 75.00,
  "recommendations": [
    {
      "recommendation_id": "rec-xyz789",
      "recommendation_type": "right_sizing",
      "resource_id": "i-1234567890abcdef0",
      "description": "Right-size resource to match utilization",
      "estimated_savings": 75.00,
```

```
      "confidence_score": 0.85,
      "implementation_steps": [...]
    }
  ],
  "summary": "Cost Optimization Analysis Summary...",
  "workflow_status": "complete"
}
```

Workflow State Management

The workflow uses TypedDict for state management:
- **Resources**: Input cloud resources to analyze
- **Analysis Results**: Detected waste and inefficiencies
- **Recommendations**: Generated optimization actions
- **Summary**: Executive summary of findings

Future Enhancements

- LLM integration for intelligent recommendations
- Conditional workflow branching
- Workflow persistence and history
- Multi-agent coordination

✅ VALIDATION COMMANDS

Step 1: Install New Dependencies

bash

```bash
cd services/cost-agent
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install LangGraph and dependencies
pip install -r requirements.txt

# Verify LangGraph installation
python -c "import langgraph; print(langgraph.__version__)"
# Expected: 0.0.20

python -c "import langchain; print(langchain.__version__)"
# Expected: 0.1.0
```

Step 2: Run Tests

bash

```bash
# Run all tests (should now be 18+ tests)
pytest

# Expected output:
# ===== 18 passed in 1.50s =====
# (8 existing + 7 workflow + 6 API tests)

# Run with coverage
pytest --cov=src --cov-report=term-missing

# Expected: Coverage >80%
```

Step 3: Start Server and Test New Endpoint

bash

```
# Terminal 1: Start server
python src/main.py

# Expected output:
# {"levelname":"INFO","message":"LangGraph workflow initialized"}
# INFO:     Uvicorn running on http://0.0.0.0:8001
```

Step 4: Test Analysis Endpoint

bash

```bash
# Terminal 2: Test the new /analyze endpoint

# Test with sample data
curl -X POST http://localhost:8001/analyze \
  -H "Content-Type: application/json" \
  -d '{
    "resources": [
      {
        "resource_id": "i-1234567890abcdef0",
        "resource_type": "ec2",
        "provider": "aws",
        "region": "us-east-1",
        "cost_per_month": 150.00,
        "utilization": 0.25,
        "tags": {"environment": "production"}
      }
    ]
  }'

# Expected response (200 OK):
# {
#   "request_id": "req-abc123",
#   "timestamp": "2025-10-18T10:00:00Z",
#   "resources_analyzed": 1,
#   "total_waste_detected": 75.00,
#   "total_potential_savings": 75.00,
#   "recommendations": [...],
#   "summary": "Cost Optimization Analysis Summary...",
#   "workflow_status": "complete"
# }
```

Step 5: Verify API Documentation

bash

```
# Open Swagger UI in browser
# http://localhost:8001/docs

# Should see:
# - GET /health (existing)
# - GET / (existing)
# - POST /analyze (NEW)

# Test the endpoint interactively in Swagger UI
```

Step 6: Run Code Quality Checks

bash

```
# Format new code
black src/ tests/
# Expected: All done! ✨

# Lint
flake8 src/ tests/ --max-line-length=88
# Expected: No errors

# Type check (optional)
mypy src/
# Expected: Success: no issues found
```

Step 7: Test Workflow Visualization (Optional)

bash

```
# Test workflow can be created
python -c "
from src.workflows.cost_optimization import create_cost_optimization_workflow
workflow = create_cost_optimization_workflow()
print('Workflow created successfully')
print('Workflow type:', type(workflow))
"

# Expected output showing workflow was created
```

Step 8: Git Commit

bash

```
git add .
git commit -m "PILOT-04: LangGraph integration complete ✅

- LangGraph workflow with 3 nodes (analyze, recommend, summarize)
- POST /analyze endpoint with full request/response models
- State management with TypedDict
- 18+ tests passing (8 existing + 10 new)
- 80%+ test coverage maintained
- Full API documentation updated"

git push
```

🎯 SUCCESS CRITERIA CHECKLIST

After running all validation commands, verify:

Dependencies

- [ ] LangGraph 0.0.20 installed
- [ ] LangChain 0.1.0 installed
- [ ] networkx installed
- [ ] matplotlib installed
- [ ] All imports working

Workflow

- [ ] 3 nodes created (analyze, recommend, summarize)
- [ ] StateGraph compiles successfully
- [ ] Workflow executes end-to-end
- [ ] State persists between nodes
- [ ] Workflow returns complete state

API

- [ ] POST /analyze endpoint works
- [ ] Returns valid JSON response
- [ ] Swagger docs updated
- [ ] Request validation works
- [ ] Error handling works

Testing

- [ ] 18+ tests passing
- [ ] Test coverage >80%
- [ ] No test failures
- [ ] Workflow tests cover all nodes
- [ ] API tests cover error cases

Code Quality

- [ ] Black formatted
- [ ] Flake8 clean
- [ ] Type hints present
- [ ] Documentation updated
- [ ] Git committed

Expected Time: < 50 minutes total (30 min generation + 20 min verification)

🚨 TROUBLESHOOTING

Issue 1: LangGraph import errors

Error: `ModuleNotFoundError: No module named 'langgraph'`

Solution:


bash

```bash
# Make sure you're in virtual environment
which python  # Should show venv path

# Reinstall dependencies
pip install --upgrade pip
pip install -r requirements.txt

# Verify installation
pip list | grep langgraph
```

Issue 2: Workflow execution hangs

Error: Workflow never completes or times out

Solution:


bash

```bash
# Check workflow structure
python -c "
from src.workflows.cost_optimization import create_cost_optimization_workflow
workflow = create_cost_optimization_workflow()
print('Workflow created')
"

# Add timeout to workflow execution (in analyze.py)
config = RunnableConfig(
    run_name=f"cost_analysis_{request_id}",
    timeout=60  # Add this
)
```

Issue 3: State not persisting

Error: Nodes don't see data from previous nodes

Solution:


python

```
# Verify state structure matches TypedDict
# In each node, return complete state:

def analyze_resources(state: CostOptimizationState) -> Dict[str, Any]:
    return {
        **state,  # IMPORTANT: Spread existing state
        "analysis_results": new_results,  # Add new fields
        "workflow_status": "analyzing",
    }
```

Issue 4: Tests fail after adding LangGraph

Error: Existing tests start failing

Solution:

bash

```
# Check if imports changed
# Verify all __init__.py files exist
ls src/workflows/__init__.py
ls src/nodes/__init__.py

# Run tests with verbose output
pytest -v
```

Issue 5: API endpoint returns 500 error

Error: POST /analyze returns Internal Server Error

Solution:

bash

```
# Check server logs for detailed error
# Test workflow separately
python -c "
from src.workflows.cost_optimization import cost_optimization_workflow
from datetime import datetime

state = {
    'resources': [],
    'request_id': 'test',
    'timestamp': datetime.utcnow(),
    'analysis_results': None,
    'total_waste_detected': 0.0,
    'recommendations': None,
    'total_potential_savings': 0.0,
    'summary': None,
    'workflow_status': 'pending',
    'error_message': None,
}

result = cost_optimization_workflow.invoke(state)
print('Success:', result['workflow_status'])
"
```

Issue 6: Version compatibility issues

Error: LangGraph/LangChain API has changed

Solution:

bash

```
# Check installed versions
pip list | grep lang

# If using newer versions, update imports:
# LangGraph 0.1.x+ might have different API
# Check: https://python.langchain.com/docs/langgraph
```

📦 DELIVERABLES

This prompt should generate:

1. New Python Source Files (9 files):

- `src/workflows/__init__.py` - Package init
- `src/workflows/state.py` - State definitions (~80 lines)
- `src/workflows/cost_optimization.py` - Main workflow (~80 lines)
- `src/nodes/__init__.py` - Package init
- `src/nodes/analyze.py` - Analysis node (~70 lines)
- `src/nodes/recommend.py` - Recommendation node (~70 lines)
- `src/nodes/summarize.py` - Summary node (~50 lines)
- `src/models/analysis.py` - API models (~120 lines)
- `src/api/analyze.py` - Analyze endpoint (~90 lines)

2. Modified Files (3 files):

- `src/main.py` - Added /analyze endpoint integration
- `src/config.py` - Added LangGraph settings
- `tests/conftest.py` - Added workflow fixtures

3. New Test Files (2 files):

- `tests/test_workflow.py` - 7+ workflow tests (~150 lines)
- `tests/test_analyze_api.py` - 6+ API tests (~120 lines)

4. Updated Configuration:

- `requirements.txt` - Added LangGraph dependencies
- `README.md` - Added LangGraph documentation

5. Working LangGraph Integration:

- 3-node workflow (analyze → recommend → summarize)
- State management with TypedDict
- POST /analyze endpoint
- 18+ tests passing
- 80%+ coverage maintained
- Full API documentation

🎯 NEXT STEPS

After this prompt succeeds:

1. ✅ **Verify:** Server running, /analyze endpoint works, all tests pass
2. ✅ **Test:** Send sample request to /analyze, verify response
3. ✅ **Commit:** `git add . && git commit -m "PILOT-04: LangGraph integration"`
4. ➡️ **Continue:** PILOT-05 (Multi-Agent Communication) or Week 1 Phase

What we'll add later:

- **Week 1:** Real LLM integration (OpenAI/Anthropic)
- **Week 1:** Cloud provider collectors (AWS/GCP/Azure)
- **Week 2:** Advanced workflow patterns (conditional branching)
- **Week 2:** Workflow persistence and history
- **Week 3:** Multi-agent coordination

📝 NOTES FOR WINDSURF

IMPORTANT INSTRUCTIONS:

1. Use LangGraph Best Practices:

- ✅ Use `StateGraph` not legacy `Graph`
- ✅ Use `TypedDict` for state (not Pydantic models)
- ✅ Return complete state from each node (`{...state, new_field: value}`)
- ✅ Use `add_edge` for deterministic flow
- ✅ Use `END` to terminate workflow

2. State Management:

- ✅ Always spread existing state in node returns
- ✅ Use Optional types for fields populated later
- ✅ Initialize all state fields in initial_state
- ❌ Don't mutate state directly

3. Code Quality:

- ✅ Type hints on all functions
- ✅ Async/await for FastAPI endpoints
- ✅ Comprehensive error handling
- ✅ Structured logging with context
- ✅ Pydantic v2 models for API

4. Testing:

- ✅ Test workflow creation
- ✅ Test individual nodes
- ✅ Test end-to-end workflow execution
- ✅ Test API endpoint with fixtures
- ✅ Test error cases

5. DO NOT:

- ❌ Use Pydantic models for state (use TypedDict)
- ❌ Use deprecated LangGraph APIs
- ❌ Skip error handling in workflow nodes
- ❌ Forget to compile the graph
- ❌ Make breaking changes to existing endpoints

SPECIAL NOTES:

- LangGraph version 0.0.20 is used (early version)
- Key APIs: StateGraph, add_node, add_edge, set_entry_point, compile, invoke
- Future versions (0.1.x+) may have different syntax
- All workflow nodes must return complete state (spread operator)

EXECUTE ALL TASKS. CREATE COMPLETE, WORKING LANGGRAPH INTEGRATION. THIS PROVES WINDSURF CAN HANDLE AI WORKFLOW ORCHESTRATION.": analysis_results, "total_waste_detected": total_waste, "workflow_status": "analyzing", }

7. CREATE src/nodes/recommend.py (Recommendation Node)
```python
"""
Recommendation node - Generates cost optimization recommendations.
"""
import logging
import uuid
from typing import Dict, Any

from src.workflows.state import CostOptimizationState, Recommendation

logger = logging.getLogger("cost_agent")


def generate_recommendations(state: CostOptimizationState) -> Dict[str, Any]:
    """
    Generate actionable cost optimization recommendations.

    This is simplified. In production, this would:
    - Use LLM to generate detailed recommendations
    - Consider business context and constraints
    - Prioritize by ROI and implementation complexity

    Args:
        state: Current workflow state with analysis results

    Returns:
        Updated state with recommendations
    """
    logger.info("Generating recommendations")

    recommendations = []
    total_savings = 0.0

    for i, analysis in enumerate(state.get("analysis_results", [])):
        if analysis["waste_detected"]:
            resource_id = analysis["metrics"]["resource_id"]
            waste_amount = analysis["waste_amount"]

            # Generate recommendation based on waste pattern
```

```
            rec: Recommendation = {
                "recommendation_id": str(uuid.uuid4()),
                "recommendation_type": "right_sizing",
                "resource_id": resource_id,
                "description": f"Right-size resource {resource_id} to match utilization",
                "estimated_savings": waste_amount,
                "confidence_score": 0.85,
                "implementation_steps": [
                    f"1. Analyze workload patterns for {resource_id}",
                    "2. Identify appropriate smaller instance size",
                    "3. Schedule downtime window",
                    "4. Resize instance",
                    "5. Monitor performance for 24 hours",
                ],
            }

            recommendations.append(rec)
            total_savings += waste_amount

    logger.info(
        f"Generated {len(recommendations)} recommendations. "
        f"Total potential savings: ${total_savings:.2f}/month"
    )

    return {
        **state,
        "recommendations": recommendations,
        "total_potential_savings": total_savings,
        "workflow_status": "recommending",
    }
```

8. CREATE src/nodes/summarize.py (Summary Node)
```python
"""
Summary node - Creates executive summary of analysis and recommendations.
"""
import logging
from typing import Dict, Any

from src.workflows.state import CostOptimizationState
```

```
logger = logging.getLogger("cost_agent")


def create_summary(state: CostOptimizationState) -> Dict[str, Any]:
    """
    Create an executive summary of the cost optimization analysis.

    In production, this would use an LLM to generate a natural language summary.

    Args:
        state: Current workflow state

    Returns:
        Updated state with summary
    """
    logger.info("Creating summary")

    num_resources = len(state["resources"])
    num_recommendations = len(state.get("recommendations", []))
    total_waste = state.get("total_waste_detected", 0.0)
    total_savings = state.get("total_potential_savings", 0.0)

    summary = f"""
Cost Optimization Analysis Summary
==================================

Resources Analyzed: {num_resources}
Waste Detected: ${total_waste:.2f}/month
Recommendations: {num_recommendations}
Potential Savings: ${total_savings:.2f}/month

Key Findings:
- {num_recommendations} optimization opportunities identified
- Average savings potential: ${total_savings/num_recommendations if num_recommendations > 0 else 0:.2f} per recommendation
- Estimated ROI implementation time: 2-4 weeks

Next Steps:
1. Review recommendations with stakeholders
2. Prioritize by business impact
```

3. Create implementation plan
4. Schedule execution windows
""".strip()

    logger.info("Summary created")

    return {
        **state,
        "summary": summary,
        "workflow_status": "complete",
    }
```

### 9. CREATE src/workflows/cost_optimization.py (Main Workflow)
```python
"""
Main LangGraph workflow for cost optimization.
"""
import logging
from typing import Dict, Any

from langgraph.graph import StateGraph, END
from langchain_core.runnables import RunnableConfig

from src.workflows.state import CostOptimizationState
from src.nodes.analyze import analyze_resources
from src.nodes.recommend import generate_recommendations
from src.nodes.summarize import create_summary

logger = logging.getLogger("cost_agent")


def create_cost_optimization_workflow() -> StateGraph:
    """
    Create the cost optimization workflow graph.

    Flow:
    START → analyze → recommend → summarize → END

    Returns:
        Compiled StateGraph ready for execution

```
    """
    # Create the graph with our state type
    workflow = StateGraph(CostOptimizationState)

    # Add nodes
    workflow.add_node("analyze", analyze_resources)
    workflow.add_node("recommend", generate_recommendations)
    workflow.add_node("summarize", create_summary)

    # Define the flow
    workflow.set_entry_point("analyze")
    workflow.add_edge("analyze", "recommend")
    workflow.add_edge("recommend", "summarize")
    workflow.add_edge("summarize", END)

    # Compile the graph
    app = workflow.compile()

    logger.info("Cost optimization workflow created")

    return app


def visualize_workflow(workflow: StateGraph, output_path: str = "workflow.png"):
    """
    Visualize the workflow graph (optional, requires matplotlib).

    Args:
        workflow: The workflow to visualize
        output_path: Where to save the visualization
    """
    try:
        import matplotlib.pyplot as plt
        from langgraph.graph import draw_mermaid

        # This is a placeholder - actual implementation depends on LangGraph version
        logger.info(f"Workflow visualization would be saved to {output_path}")
        # In practice: draw_mermaid(workflow).render(output_path)

    except ImportError:
        logger.warning("Matplotlib not available, skipping visualization")
```

```
# Create a singleton instance
cost_optimization_workflow = create_cost_optimization_workflow()
```

10. CREATE src/models/analysis.py (API Models)
```python
"""
Request/response models for analysis endpoint.
"""
from typing import List, Dict, Any, Optional
from datetime import datetime
from pydantic import BaseModel, Field


class ResourceRequest(BaseModel):
    """Single resource to analyze"""
    resource_id: str = Field(..., description="Unique resource identifier")
    resource_type: str = Field(..., description="Type of resource (e.g., 'ec2', 'rds')")
    provider: str = Field(..., description="Cloud provider (e.g., 'aws', 'gcp', 'azure')")
    region: str = Field(..., description="Cloud region")
    cost_per_month: float = Field(..., ge=0, description="Monthly cost in USD")
    utilization: float = Field(..., ge=0, le=1, description="Utilization percentage (0-1)")
    tags: Dict[str, str] = Field(default_factory=dict, description="Resource tags")

    class Config:
        json_schema_extra = {
            "example": {
                "resource_id": "i-1234567890abcdef0",
                "resource_type": "ec2",
                "provider": "aws",
                "region": "us-east-1",
                "cost_per_month": 150.00,
                "utilization": 0.25,
                "tags": {"environment": "production", "team": "backend"},
            }
        }


class AnalysisRequest(BaseModel):
```

```
    """Request to analyze resources for cost optimization"""
    resources: List[ResourceRequest] = Field(..., min_length=1, description="Resources to analyze")

    class Config:
        json_schema_extra = {
            "example": {
                "resources": [
                    {
                        "resource_id": "i-1234567890abcdef0",
                        "resource_type": "ec2",
                        "provider": "aws",
                        "region": "us-east-1",
                        "cost_per_month": 150.00,
                        "utilization": 0.25,
                        "tags": {"environment": "production"},
                    }
                ]
            }
        }


class RecommendationResponse(BaseModel):
    """Single optimization recommendation"""
    recommendation_id: str
    recommendation_type: str
    resource_id: str
    description: str
    estimated_savings: float
    confidence_score: float
    implementation_steps: List[str]


class AnalysisResponse(BaseModel):
    """Response from cost optimization analysis"""
    request_id: str = Field(..., description="Unique request identifier")
    timestamp: datetime = Field(..., description="Analysis timestamp")
    resources_analyzed: int = Field(..., description="Number of resources analyzed")
    total_waste_detected: float = Field(..., description="Total waste in USD/month")
    total_potential_savings: float = Field(..., description="Total potential savings in USD/month")
    recommendations: List[RecommendationResponse] = Field(..., description="Optimization recommendations")
    summary: str = Field(..., description="Executive summary")
```

```
    workflow_status: str = Field(..., description="Workflow status")

    class Config:
        json_schema_extra = {
            "example": {
                "request_id": "req-abc123",
                "timestamp": "2025-10-18T10:00:00Z",
                "resources_analyzed": 5,
                "total_waste_detected": 375.00,
                "total_potential_savings": 375.00,
                "recommendations": [
                    {
                        "recommendation_id": "rec-xyz789",
                        "recommendation_type": "right_sizing",
                        "resource_id": "i-1234567890abcdef0",
                        "description": "Right-size instance to match utilization",
                        "estimated_savings": 75.00,
                        "confidence_score": 0.85,
                        "implementation_steps": ["1. Analyze workload", "2. Resize instance"],
                    }
                ],
                "summary": "Found 3 optimization opportunities...",
                "workflow_status": "complete",
            }
        }
```

11. CREATE src/api/analyze.py (API Endpoint)

```python
"""
Analysis endpoint for cost optimization.
"""
import logging
import uuid
from datetime import datetime

from fastapi import APIRouter, HTTPException
from langchain_core.runnables import RunnableConfig

from src.models.analysis import AnalysisRequest, AnalysisResponse, RecommendationResponse
from src.workflows.cost_optimization import cost_optimization_workflow
```

```
from src.workflows.state import CostOptimizationState, ResourceInfo

router = APIRouter()
logger = logging.getLogger("cost_agent")



@router.post("/analyze", response_model=AnalysisResponse)
async def analyze_costs(request: AnalysisRequest) -> AnalysisResponse:
    """
    Analyze resources for cost optimization opportunities.

    This endpoint runs the LangGraph workflow to:
    1. Analyze resources for waste
    2. Generate recommendations
    3. Create executive summary

    Args:
        request: Analysis request with resources to analyze

    Returns:
        AnalysisResponse: Complete analysis with recommendations
    """
    request_id = f"req-{uuid.uuid4().hex[:8]}"
    logger.info(f"Starting cost analysis {request_id} for {len(request.resources)} resources")

    try:
        # Convert request to workflow state
        resources: list[ResourceInfo] = [
            {
                "resource_id": r.resource_id,
                "resource_type": r.resource_type,
                "provider": r.provider,
                "region": r.region,
                "cost_per_month": r.cost_per_month,
                "utilization": r.utilization,
                "tags": r.tags,
            }
            for r in request.resources
        ]

        initial_state: CostOptimizationState = {
```

```
        "resources": resources,
        "request_id": request_id,
        "timestamp": datetime.utcnow(),
        "analysis_results": None,
        "total_waste_detected": 0.0,
        "recommendations": None,
        "total_potential_savings": 0.0,
        "summary": None,
        "workflow_status": "pending",
        "error_message": None,
    }

    # Run the workflow
    config = RunnableConfig(run_name=f"cost_analysis_{request_id}")
    result = cost_optimization_workflow.invoke(initial_state, config)

    # Convert workflow result to API response
    recommendations = [
        RecommendationResponse(**rec) for rec in result.get("recommendations", [])
    ]

    response = AnalysisResponse(
        request_id=result["request_id"],
        timestamp=result["timestamp"],
        resources_analyzed=len(result["resources"]),
        total_waste_detected=result.get("total_waste_detected", 0.0),
        total_potential_savings=result.get("total_potential_savings", 0.0),
        recommendations=recommendations,
        summary=result.get("summary", "No summary available"),
        workflow_status=result.get("workflow_status", "unknown"),
    )

    logger.info(f"Cost analysis {request_id} completed successfully")
    return response

except Exception as e:
    logger.error(f"Cost analysis {request_id} failed: {e}", exc_info=True)
    raise HTTPException(
        status_code=500,
        detail=f"Analysis failed: {str(e)}"
    )
```

```

### 12. MODIFY src/main.py (Add new endpoint)
```python
"""
OptiInfra Cost Agent - Main Application

This is the Cost Agent that optimizes cloud spending through:
- Spot instance migrations
- Reserved instance recommendations
- Instance right-sizing
"""
import asyncio
import logging
from contextlib import asynccontextmanager

from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

from src.api import health, analyze  # MODIFIED: Added analyze import
from src.config import settings
from src.core.logger import setup_logging
from src.core.registration import register_with_orchestrator

# Setup logging
logger = setup_logging()



@asynccontextmanager
async def lifespan(app: FastAPI):
    """
    Lifespan events for the FastAPI application.
    Handles startup and shutdown events.
    """
    # Startup
    logger.info("Starting OptiInfra Cost Agent")
    logger.info(f"Environment: {settings.environment}")
    logger.info(f"Port: {settings.port}")
    logger.info("LangGraph workflow initialized")  # MODIFIED: Added

    # Register with orchestrator
```
```

```
    if settings.orchestrator_url:
        try:
            await register_with_orchestrator()
            logger.info("Successfully registered with orchestrator")
        except Exception as e:
            logger.error(f"Failed to register with orchestrator: {e}")

    yield

    # Shutdown
    logger.info("Shutting down Cost Agent")


# Create FastAPI app
app = FastAPI(
    title="OptiInfra Cost Agent",
    description="AI-powered cost optimization agent with LangGraph workflows",  # MODIFIED
    version="0.2.0",  # MODIFIED: Version bump
    lifespan=lifespan,
)

# Add CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Include routers
app.include_router(health.router, tags=["health"])
app.include_router(analyze.router, tags=["analysis"])  # MODIFIED: Added


# Root endpoint
@app.get("/")
async def root():
    """Root endpoint - service information"""
    return {
        "service": "OptiInfra Cost Agent",
```

```
        "version": "0.2.0",  # MODIFIED: Version bump
        "status": "running",
        "capabilities": [
            "spot_migration",
            "reserved_instances",
            "right_sizing",
            "ai_workflow_optimization",  # MODIFIED: Added
        ],
    }


# Run with uvicorn when executed directly
if __name__ == "__main__":
    import uvicorn

    uvicorn.run(
        "src.main:app",
        host="0.0.0.0",
        port=settings.port,
        reload=settings.environment == "development",
        log_level=settings.log_level.lower(),
    )
```

13. CREATE tests/test_workflow.py (Workflow Tests)
```python
"""
Tests for LangGraph workflow.
"""
import pytest
from datetime import datetime

from src.workflows.cost_optimization import create_cost_optimization_workflow
from src.workflows.state import CostOptimizationState, ResourceInfo


@pytest.fixture
def sample_resources():
    """Sample resources for testing"""
    return [
        {
```

```
        "resource_id": "i-test001",
        "resource_type": "ec2",
        "provider": "aws",
        "region": "us-east-1",
        "cost_per_month": 100.0,
        "utilization": 0.2,  # Low utilization - should trigger recommendation
        "tags": {"env": "test"},
    },
    {
        "resource_id": "i-test002",
        "resource_type": "ec2",
        "provider": "aws",
        "region": "us-west-2",
        "cost_per_month": 200.0,
        "utilization": 0.8,  # High utilization - no recommendation
        "tags": {"env": "prod"},
    },
]


@pytest.fixture
def initial_state(sample_resources):
    """Create initial workflow state"""
    return {
        "resources": sample_resources,
        "request_id": "test-req-001",
        "timestamp": datetime.utcnow(),
        "analysis_results
```